Implementation of Twins and Testing of diemBFT

Test Generator:

Config file containing the test parameters for the test generator contains the following:

```
   "N_validators",  // Number of validators
   "N_partitions",  // Number of non-empty partitions
   "Allowed_leader_type", // Type of leader, “FAULTY”, “NON-FAULTY”, “ALL”.
   "N_rounds", // Number of rounds.
   "is_deterministic", // Whether to generate test cases sequentially or
randomized.
   "Is_with_replacement",  // Whether to enumerate with replacement when permuting
scenarios over n_rounds.
   // Only used when is_deterministic is False
   "N_testcases", // Number of test cases to be generated.
   "Test_file_batch_size", // Number of test-case in each test case file.
   "N_twins",  // Number of twins.
   "Generate_valid_partition", // To ensure progress
   "Intra_partition_drop_types",   // Subset of {Proposal, Vote, Timeout}, list of
message types that can be dropped intra-partition
   // We allow a maximum of 2 message types to be dropped. If more than 2 are
given, we take the first 2 in the list
   "seed": 12345
```

Helper functions for each step of the test generator are as follows:

- **Step 1:** get_all_possible_partitions()
- **Step 2:** generate_leader_partitions()
    - Step 2.1 : enumerate_leader_partitions_with_drops()
- **Step 3:** enumerate_leader_partition_pairs_over_rounds()

STEP - 1:

Generate all possible partition scenarios based on the flag “generate_valid_partitions” to ensure the progress.

```
// Returns all possible ways the list of validator IDs can be divided into
n_partitions.
//"generate valid partition" indicates that the generated partitions will
eventually assure the presence of quorum.
Procedure get_all_possible_partitions(validator_ids, n_partitions,
generate_valid_partition, num_faulty):
```

```
   all_possible_partitions <- []
   t = int(count_part(len(validator_ids), n_partitions))
   for i in range(t):
       x <- gen_part(validator_ids, n_partitions, i)
       if generate_valid_partition:
       // isValidPartition() will check if in the generated partition quorum can
exist.
           if isValidPartition(x, num_faulty):
               all_possible_partitions.append(x)
       else:
           all_possible_partitions.append(x)
   return all_possible_partitions
```

STEP - 2:

Generate all the possible leader-partition pairs.

Args:

- all the partition scenarios,
- validator ids: id of all the validators including twins,
- twin_ids: id of all twin validators
- leader_type: variable to control the type of leader in each round.

```
Procedure generate_leader_partitions(partition_scenarios, validator_ids, twin_ids,
leader_type):
   leader_partitions_pairs <- []

//contains a list of all validator ids except twins ids.
   all_validator_ids <- set()
   [all_validator_ids.update(partition) for partition in partition_scenarios[0]]

//faulty_validators_id_list contains faulty validator id.
   faulty_validators_id_list <- [validatorId.split("_")[0] for validatorId in
all_validator_ids if len(validatorId.split("_")) > 1]

// Assigning leader in each subset of partition depending on the type of leader.
   for partition in partition_scenarios:
       for validator_id in validator_ids:
           if leader_type == "ALL":
               leader_partitions_pairs.append((validator_id, partition))
           elif leader_type == "FAULTY" and validator_id in twin_ids:
               leader_partitions_pairs.append((validator_id, partition))
           elif leader_type == "NON-FAULTY" and validator_id not in
faulty_validators_id_list:
               leader_partitions_pairs.append((validator_id, partition))
```

```
   return leader_partitions_pairs
```

STEP - 2.1:
Combine leader-partition pairs with every kind of intra-partition message drop type i.e. Vote, Proposal, or Timeout.

```
Procedure enumerate_leader_partitions_with_drops(leader_partitions, drop_types):
   drop_types <- drop_types[:2]
   result <- []
   drop_scenarios <- [[]]

   // no drop scenario.
   if len(drop_types) > 0:
       drop_scenarios.append(drop_types)

   // generate all the possible drop combinations
   if len(drop_types) > 1:
       drop_scenarios += [[drop_type] for drop_type in drop_types]

   for leader, partition in leader_partitions:
       for drop_scenario in drop_scenarios:
           result.append((leader, partition, drop_scenario))
   return result
```

STEP - 3:
Combine rounds with leader-partition pairs with or without replacement and write the required number of test cases in a file in a batch of size “batch_size”.

```
Procedure enumerate_leader_partition_pairs_over_rounds(leader_partition_pairs,
n_rounds, n_testcases, is_deterministic,is_with_replacement, validator_twin_ids,
n_validators, batch_size):

   round_leader_partition_pairs <- []
   total_leader_partition <- len(leader_partition_pairs)
   index_list <- list(range(total_leader_partition))

   // Combine rounds with leader partition pair in a deterministic way without
replacement.
   if not is_with_replacement and is_deterministic:
       all_round_combinations <- []

       for each_combination in list(itertools.combinations(index_list, n_rounds)):
```

```
            all_round_combinations += list(itertools.permutations(each_combination))
            if len(all_round_combinations) > n_testcases:
                break

    else:
        if is_deterministic:
            permutations <- [[] for _ in range(total_leader_partition ** n_rounds)]
            all_round_combinations <-
permutations_with_replacement(total_leader_partition, n_rounds,

permutations)
        else:
            all_round_combinations <- enumerate_randomized(leader_partition_pairs,
n_rounds, n_testcases)

    count_testcases <- 0
    flag <- False

    for permutation in all_round_combinations:
        round_leader_partition_pairs.append(
            accumulate(permutation, leader_partition_pairs, n_rounds,
validator_twin_ids, n_validators))
        count_testcases += 1

        if count_testcases == n_testcases:
            flag <- True

        if not flag and len(round_leader_partition_pairs) == batch_size:
            dump_file(round_leader_partition_pairs, count_testcases)
            round_leader_partition_pairs <- []

        elif flag and round_leader_partition_pairs:
            dump_file(round_leader_partition_pairs, count_testcases)
            break


// Enumerate n_test_cases ways of randomly arranging leader_partition pairs over
n_rounds
Procedure enumerate_randomized(leader_partition_pairs, n_rounds, n_test_cases):
    return [[random.randrange(len(leader_partition_pairs)) for _ in range(n_rounds)]
for _ in range(n_test_cases)]


// Return all ways of arranging n leader_partition pairs over k rounds
Procedure permutations_with_replacement(n, k, permutations):
    m <- 0
    if k < 1:
```

```
        return permutations

    for i in range(27):
        permutations[i].append(m % n)
        if (i % n ** (k - 1)) == n ** (k - 1) - 1:
            m <- m + 1

    return permutations_with_replacement(n, k - 1, permutations)

// Convert given testcase into a JSON object
Procedure accumulate(index_list, leader_partition_pairs, n_rounds,
validator_twin_ids, n_validators):
    round_leader_partitions <- [leader_partition_pairs[idx] for idx in index_list]
    curr_test_case <- JsonObject(n_validators, n_rounds, validator_twin_ids,
round_leader_partitions)

    return curr_test_case.toJSON()

// Writes the list of JSON elements to file
Procedure dump_file(elements, file_count):
    file_name: str <- "../testcases/testcases_batch_" + str(file_count) + ".jsonl"

    with open(file_name, "wb") as outfile:
        for element in elements:
            if element is not None:
                outfile.write(element.encode() + b"\n")
```

Execution point of test generator.

```
if 'seed' in generator_config:
random.seed(generator_config['seed'])

 // Generate all the validator ids.
 validator_ids <- [validator_id for validator_id in
range(generator_config['n_validators'])]

// Generate all twin validator ids based on the number of validators.
 twin_ids <- [validator_id for validator_id in range(generator_config['n_twins'])]

 validator_and_twin_ids <- [str(validator_id) for validator_id in validator_ids] +
[str(validator_id) + "_twin" for validator_id in twin_ids]

    // Step 1: generate all possible partition scenarios.
 partition_scenarios <- generate_partitions.get_all_possible_partitions(
```

```
   validator_and_twin_ids, generator_config['n_partitions'])

   // Step 2: generate all possible leader-partition pairs.
 leader_partition_pairs <- generate_leader_partitions(
       partition_scenarios, validator_ids, twin_ids,
generator_config['allowed_leader_type'])

  // Step 2.1: Adding message drops
 leader_partitions_with_drops <- enumerate_leader_partitions_with_drops(
       leader_partition_pairs, generator_config['intra_partition_drop_types'])

   // Step 3: generate test-cases by combining rounds with leader partition pairs
in either deterministic or randomized ways.

enumerate_leader_partition_pairs_over_rounds(leader_partition_pairs=leader_partitio
ns_with_drops, n_rounds=generator_config['n_rounds'],
n_testcases=generator_config['n_testcases'],
is_deterministic=generator_config['is_deterministic'],
is_with_replacement=generator_config['is_with_replacement'],
validator_twin_ids=twin_ids, n_validators=len(validator_ids),
batch_size=generator_config['test_file_batch_size'])
```

Test Executor :

The Test Executor is the module with which the user primarily interacts in order to run test cases. The user passes a list of TestConfig objects as a JSON file. Each TestConfig object describes a test case that needs to be executed by the DiemTwins testing framework.

Design assumptions

Here, we briefly list out the design decisions/assumptions taken in addition to the ones presented in the Twins paper and the phase 3 document.

- We let each validator run for n_rounds + 3 rounds. Here, the first n_rounds will be simulated as per the TestCase generated by our Test Generator module. We will then allow the system to execute 3 more rounds without any network partitions. This will allow all non-faulty replicas to commit any pending blocks, and bring their ledgers up to date.

```
class TestExecutor(process):
   Procedure setup(test_case, test_id, test_file_id):
       self.n_rounds <-  test_case.n_rounds
       self.n_validators <-  test_case.n_validators
```

```
        self.n_faulty <-  (self.n_validators - 1)//3
        self.delta <-  test_case.delta
        self.config_id <-  str(test_file_id) + "_" + str(test_id)

    Procedure run():
        start_time <-  time.time()
        private_keys_validators <-  []
        public_keys_validators <-  []

        for validator_id in range(self.n_validators):
            private_key, public_key <-  Cryptography.generate_key()
            private_keys_validators.append(private_key)
            public_keys_validators.append(public_key)

        for validator_id in range(self.n_validators):
            if validator_id not in test_case.twin_ids:
                Procedure ault_leader <-  validator_id
                break

        validator_config <-  {
            "config_id": config_id,
            "nrounds": self.n_rounds,
            "nvalidators": self.n_validators,
            "nfaulty": self.n_faulty,
            "delta": self.delta,
            "leaders": [leader_partition.leader for leader_partition in
test_case.leader_partitions],
            "Procedure ault_leader": Procedure ault_leader
        }

        validator_map <-  dict()

        for validator_id in range(self.n_validators):
            // Create a validator and add to validator_map. If its not a   twin, its
twin_id is simply the validator_id
            validator <-  new(ValidatorTwins, num=1)
            twin_id <-  str(validator_id)
            validator_map[twin_id] <-  validator

            if validator_id in test_case.twin_ids:
                // Create the twin validator process and also add it to the
validator map
                validator <-  new(ValidatorTwins, num=1)
                twin_id <-  str(validator_id) + "_twin"
                validator_map[twin_id] <-  validator

        all_validators <-  set()
```

```
       for twin_id, validator in validator_map.items():
           validator_id <-  int(twin_id[0])
           setup(
               validator,
               (validator_config, validator_id,
private_keys_validators[validator_id],
                public_keys_validators, twin_id, test_case, validator_map))
           all_validators <-  all_validators.union(validator)

       start(all_validators)
       await(each(v in all_validators, has=received(('Done',), from_=v)))
       twin_ids <-  [twin_id for twin_id in validator_map]

// Orchestrator is to maintain all the executor processes to run multiple test
cases.
class Orchestrator(process):
     Procedure setup():
        Pass

     Procedure run():
       start_time <-  time.time()

       // Get list of all test files
       test_files <-  sorted(glob.glob(self.test_directory + "*.jsonl"), key=lambda
file_name: (len(file_name), file_name))

       // Read and execute each test file one at a time
       for test_file_no, test_file_name in enumerate(test_files):
            with open(test_file_name, 'r') as test_file:
               test_cases <-  [TestCase(
                       n_rounds=json_obj.n_rounds,
                       n_validators=json_obj.n_validators,
                       leader_partitions=[LeaderPartition(round_leader_partition)
for round_leader_partition in
                                                json_obj.round_leader_partitions],
                       twin_ids=json_obj.twin_ids,
                       delta=0.1
                   )
                   for json_obj in [
                       json.loads(json_string,
object_hook=JsonObject.object_decoder)
                       for json_string in test_file.readlines()]]

           Executors <- set()

           for test_id, test_case in enumerate(test_cases):
               executor <-  new(TestExecutor)
```

```
                setup(executor, (test_case, test_id, test_file_no))
                start(executor)
                executors.add(executor)

            await(each(executor in executors, has=received(('Done',),
from_=executor)))
            time.sleep(1)

Procedure  main():
   orchestrator <- new(Orchestrator)
   setup(orchestrator, ("../testcases/",))
   start(orchestrator)
```

Safety Module: To ensure safety property, check if the orders of the transactions committed in the ledger file of all the validators are the same or not. If it's matching, safety property is ensured else not.

```
 Procedure check_safety(validator_ids):
       for validator_id_a in validator_ids:
          for validator_id_b in validator_ids:
                for i in range(min(len(ledger_a_lines), len(ledger_b_lines))):
                   if ledger_a_lines[i] != ledger_b_lines[i]:
                       return False
       return True
```

Liveness Module: To ensure the liveness property a "no-op" transaction is sent at the end of all the transactions. If the "no-op" transaction is committed in the ledger file of all the validators, then the Liveness property is ensured.

```
 Procedure check_liveness(validator_ids):
       for validator_id in validator_ids:
           if last_ledger_line_validator != 'no-op':
               return False
       return True
```

Network playground:

```
class ValidatorTwins(process, Validator):
   Procedure send(payload, to):
       to_validator_ids <-  to
```

```
        message_type, message <-  payload

        to_validator_twin_ids  // contains the list of twin validator ids if the
destination validator id is having twin.

        if not should_partition(message_type, self.pacemaker.current_round):
            // Send the message to all recipients without applying filtering logic
            for to_validator in [validator_map[to_twin_id] for to_twin_id in
to_validator_twin_ids]:
                super().send(payload, to=to_validator)
        else:
            // Only send to the twin_ids part of the current partition
            leader_partition_scenario <-
test_case.leader_partitions[pacemaker.current_round - 1]

            current_partition <-  []
            dropped_messages <-  []

            for partition in leader_partition_scenario.partitions:
                network_partition <-  partition.partitions

                if twin_id in network_partition:
                    current_partition <-  network_partition
                    dropped_messages <-  partition.dropped_messages
                    break

            if message_type in dropped_messages:
                // Intra partition message drop
                return

            for to_validator_twin_id in to_validator_twin_ids:
                // Send to all recipients who are in the same partition
                if to_validator_twin_id in current_partition:
                    to_validator <-  validator_map[to_validator_twin_id]
                    super().send(payload, to=to_validator)
    Procedure should_partition(message_type, round_num):
        if round_num > test_case.n_rounds or message_type not in {'Proposal',
'Vote', 'Timeout'}:
            return False

        if message_type in {'Proposal', 'Vote'}:
            return True
        else:
            // Else, message type is Timeout
            // If we have not sent a timeout message for this round, return True,
else False
            ans <-  round_num not in self.timed_out_rounds
```

```
            self.timed_out_rounds.add(round_num)

            return ans
```